LA-UR-98-2249
Approved for public release; distribution is unlimited

A Computational Study of Routing Algorithms for Realistic Transportation Networks

Riko Jacob, Madhav V. Marathe, Kai Nagel

2nd Workshop on Algorithmic Engineering (WAE '98)

Saarbruecken, Germany, Aug 19-21 1998

A Computational Study of Routing Algorithms for Realistic Transportation Networks

RIKO JACOB [1]   MADHAV V. MARATHE [1]   KAI NAGEL [1]

May 28, 1998

Abstract

We carry out an experimental analysis of a number of shortest path (routing) algorithms investigated in the context of the TRANSIMS (TRansportation ANalysis and SIMulation System) project. The main focus of the paper is to study how various heuristic and exact solutions, associated data structures affected the computational performance of the software developed especially for realistic transportation networks. For this purpose we have used Dallas Ft-Worth road network with very high degree of resolution. The following general results are obtained.

1. We discuss and experimentally analyze various one-one shortest path algorithms. These include classical exact algorithms studied in the literature as well as heuristic solutions that are designed to take into account the geometric structure of the input instances.
2. We describe a number of extensions to the basic shortest path algorithm. These extensions were primarily motivated by practical problems arising in TRANSIMS and ITS (Intelligent Transportation Systems) related technologies. Extensions discussed include – (i) Time dependent networks, (ii) multi-modal networks, (iii) networks with public transportation and associated schedules.

Computational results are provided to empirically compare the efficiency of various algorithms. Our studies indicate that a modified Dijkstra's algorithm is computationally fast and an excellent candidate for use in various transportation planning applications as well as ITS related technologies.

Keywords: Experimental Analysis, Transportation Planning, Algorithms, Network Design, Shortest Paths Algorithms.

[1] Los Alamos National Laboratory, P.O. Box 1663, MS M997, Los Alamos, NM 87545. Email: {jacob, marathe, kai}@lanl.gov. Research supported by the Department of Energy under Contract W-7405-ENG-36.

1 Introduction

TRANSIMS is a multi-year project at the Los Alamos National Laboratory and is funded by the Department of Transportation and by the Environmental Protection Agency. The main purpose of TRANSIMS is to develop new methods for studying transportation planning questions. A typical example of a question that can be studied in this context would be to study the economic and social impact of building a new freeway in a large metropolitan area. We refer the reader to [TR+95a] and the web-site `http://www-transims.tsasa.lanl.gov/research_team/papers/` for more details about the TRANSIMS project.

The main goal of the paper is to describe the computational experiences in engineering various path finding algorithms specifically in the context of TRANSIMS. Most of the algorithms discussed here are not new; they have been discussed in the Operations Research and Computer Science community. Although extensive research has been done on theoretical and experimental evaluation of shortest path algorithms, most of the empirical research has focused on randomly generated networks, special classes of networks such as grids. In contrast, not much work has been done to study the computational behavior of shortest path and related routing algorithms on realistic traffic networks. The realistic networks differ with random networks as well as homogeneous (structured networks) in the following significant ways:
(i) Realistic networks typically have a very low average degree. In fact in our case the average degree of the network was around 2.6. Similar numbers have been reported in [ZN98]. In contrast random networks used in [Pa84] have in some cases average degree of up to 10.
(ii) Realistic networks are not very uniform. In fact, one typically sees one or two large clusters (downtown and neighboring areas) and then small clusters spread out throughout the entire area of interest.
(iii) For most empirical studies with random networks, the edge weights are chosen independently and uniformly at random from a given interval. In contrast, realistic networks typically have short links.

With the above reasons and specific application in mind, the main focus of this paper is to carry out experimental analysis of a number of shortest path algorithms on real transportation network and subject to practical constraints imposed by the overall system.

The rest of the report is organized as follows. Section 5 describes our experimental setup. Section 6 describes the experimental results obtained. Finally, in Section 8 we give concluding remarks and directions for future research. We have also included an Appendix (Section 8.1) that describes the relevant algorithms for finding shortest paths in detail.

2 Problem specification and justification

The problems discussed above can be formally described as follows: let $G(V, E)$ be a (un)directed graph. Each edge $e \in E$ has one attribute — $w(e)$. $w(e)$ denotes the weight of the edge (or cost) e. Here, we assume that the weights are non-negative floating point numbers. Most of our positive results can in fact be extended to handle negative edge weights also (if there are no negative cycles).

Definition 2.1 One-One Shortest Path:
Given a directed weighted, graph G, a source destination pair (s, d) find a shortest (with respect to w) path p in G from s to d.

Note that our experiments are carried out for shortest path between a pair of nodes, as against to finding shortest path trees. Much of the literature on experimental analysis uses the latter measure to gauge the efficiency. Our choice for the measure is motivated by the following observations:

1. We wanted the route planer to work for roughly a million travelers. In highly detailed networks, most of these travelers have different starting points (for example, for Portland we have 1.5 million travelers and 200 000 possible starting locations). Thus, for any given starting location, we could re-use the tree computation only for of the order of ten other travelers.

2. We wanted our algorithms to be extensible to take additional elements into account. For example, each such traveler typically has a different starting time for his/her trip. Since we use our algorithms for time dependent networks (networks in which edge weights vary with time), the shortest path tree will be different for each traveler. Another example in this context is to find paths for travelers in network with multiple mode choices. In this context, we are given a directed labeled, weighted, graph G representing a transportation network with the labels on edges representing the various modal attributes (e.g. a label t might represent a rail line). The goal is typically to find shortest (simple) paths subject to certain labeling constraints on the set of feasible paths. In general, the criteria for path selection vary so much from traveler to traveler that it becomes doubtful that the additional overhead for the "re-use" of information will pay off.

3. The TRANSIMS framework allows us to use paths that are not necessarily optimal. This motivates investigation into the possible use of heuristic solutions for obtaining near optimal paths (e.g. the modified A^* algorithm). For most of these heuristics, the idea is to bias a more focused search towards the destination – thus naturally motivating the study of one-one shortest path algorithms.

4. Finally, the networks we anticipate to deal with contain more than 80 000 nodes and around 120 000 edges. For such networks storing shortest path trees amounts to huge memory overheads.

3 Choice of algorithms

Important objectives used to evaluate the performance of the algorithms include (i) time taken for computation on real networks, (ii) quality of solution obtained, (iii) ease of implementation and (iv) extensibility of the algorithm for solving other variants of the shortest path problem. A number interesting engineering questions were encountered in the process. We experimentally evaluated a number of variants of basic Dijkstra's algorithm. The basic algorithm was chosen due to the recommendations made in Cherkassky, Goldberg and Radzik [CGR96] and Zhan and Noon [ZN98]. The algorithms studied were:

- Dijkstra's algorithm with Binary Heaps [CGR96],
- A^* algorithm proposed in AI literature and analyzed by Sedgewick and Vitter [SV86],
- a modification of the A^* algorithm that we will describe below, and alluded to in [SV86].

We also considered a bidirectional version of Dijkstra's algorithm described in [Ma, LR89]. We briefly recall the A^* algorithm and the modification proposed. Details of these algorithms can be found in the Appendix. When the underlying network is Euclidean, it is possible to improve the average case performance of Dijkstra's algorithm. Typically, while solving problems on such graphs, the inherent geometric information is ignored by the classical path finding algorithms. The basic idea of improving the performance of Dijkstra's algorithm is from Sedgewick and Vitter [SV86] and is originally attributed to Hart Nilsson and Raphel [HNR68] can be described as follows. To build a shortest path from s to t, we use the original distance estimate for the fringe vertex such as x, i.e. from s to x (as before) *plus* the Euclidean distance from x to t. Thus we use global information about the graph to guide our search for shortest path from s to t. The resulting algorithm typically runs much faster than Dijkstra's algorithm on typical graphs for the following intuitive reasons: (i) The shortest path tree grows in the direction of t and (ii) The search of the shortest path can be terminated as soon as t is added to the shortest path tree.

We can now modify this algorithm by giving an appropriate weight to to the distance from x to t. By choosing an appropriate multiplicative factor, we can increase the contribution of the second component in calculating the label of a vertex. From a intuitive standpoint this corresponds to giving the destination a high potential, in effect biasing the search towards the destination. This modification will in general **not** yield shortest paths, nevertheless our experimental results suggest that the errors produced are typically quite small.

4 Summary of Results

We are now ready to summarize the main results and conclusions of this paper. As already stated the *main focus* of the paper is towards engineering well known shortest path algorithms in a practical setting. Another goal of this paper is also to provide reasons for and against certain implementations from a practical standpoint. We believe that our conclusions along with the earlier results in [ZN98, CGR96] provide practitioners an useful basis to select appropriate algorithms/implementations in the context of transportation networks. The general results/conclusions of this paper are summarized below.

1. We conclude that the simple Binary heap implementation of Dijkstra's algorithm is a good choice for finding optimal routes in real road transportation networks. Specifically, we found that a certain types of data-structure fine tuning did not significantly improve the performance of our implementation.

2. Our results suggest that heuristic solutions that aim at using the geometric structure of the graphs are attractive candidates for future research. Our experimental results motivated the formulation and implementation of an extremely fast heuristic extension of the basic A^* algorithm that seems to yield near optimal solutions.

3. We have extended this algorithm in two orthogonal and important directions; (i) time dependent networks and (ii) multi-modal networks. These extensions are significant from a practical standpoint since they are the most realistic representations of the underlying physical network. We perform suitable tests to calculate the slow down experienced as a result of these extensions.

4. Our study suggests that bidirectional variation of Dijkstra's algorithm is not suitable for transportation planning. Our conclusions are based on two factors: (i) the algorithm is not extensible

to more general path problems and (ii) the running time of the algorithm is more than A^* algorithm.

5 Experimental Setup and Methodology

In this section we describe the computational results of our implementations. In order to anchor research in realistic problems, TRANSIMS uses example cases called *Case studies* (See [CS97] for complete details). This allows us to test the effectiveness of our algorithms on real life data. The case study just concluded was focused on Dallas Fort-Worth (DFW) Metropolitan area and was done in conjunction with Municipal Planning Organization (MPO) (known as North Central Texas Council of Governments (NCTCOG)). We generated trips for the whole DFW area for a 24 hour period. The input for each traveler has the following format: (starting time, starting location, ending location).[2] There are 10.3 million Trips over 24 hours. The number of nodes and links in the Dallas network is roughly 9863, 14750 respectively. The average degree of a node in the network was 2.5. We route all these trips through the so-called focused network. It has all freeway links, most major arterials, etc. Inside this network, there is an area where *all* streets, including local streets, are contained in the data base. This is the study area. We initially routed all trips between 5am and 10am, but only the trips which did go through the study area were retained, resulting in approx. 300 000 trips. These 300 000 trips were re-planned over and over again in iteration with the micro-simulation(s). For more details, see, e.g., [NB97, CS97]. A 3% random sample of these trips were used for our computational experiments. Finally, the number of Links of each class is as follows:

Class	Type	Number (Oct 96)	Number Feb 97
0	Centroid	2964	1422
1	Freeway	1962	1984
2	Principle Art.	2056	1251
3	Minor Art.	5079	2843
4	Collector	3830	2196
5	Local Street	144	1986
6	Freeway Ramp	2704	2124
7	Frontage Road	1037	944

Table 1: Summary of individual link types in Dallas Ft Worth Area. The third column summarizes the numbers for the network used in October 1996 study. The numbers in the fourth column summarize the numbers for the February 1997 network. As one can see the biggest change in the numbers is in the local streets. The new network used has these streets encoded and thus is used by the planner to route plans.

Preparing the network. The data received from DFW metro had a number of inadequacies from the point of view of performing the experimental analysis. These had to be corrected before carrying out the analysis. We mention a few important ones here. First, the network was found to have a number of disconnected components (small islands). We did not consider (o, d) pairs in different components. Second, a more serious problem from an algorithmic standpoint was the fact that for a number of links, the length was *less* than the actual Euclidean distance between the the two end points. In most

[2]This is roughly correct, the reality is more complicated, [NB97, CS97].

cases, this was due to an artificial convention used by the DFW transportation planners (so-called centroid connectors always have length 10 m, whatever the Euclidean distance), but in some cases it pointed to data errors. In any case, this discrepancy disallows effective implementation of A^* type algorithms. For this reason we introduce the notion of the "normalized" network: For all "too short" links we set the reported length to be equal to the Euclidean distance.

We also carried out preliminary experimental analysis for the following network modifications that could be helpful in improving the efficiency of our algorithms. These include: (i) Removing nodes with degrees less than 3: (Includes collapsing paths and also leaf nodes) (ii) Modifying nodes of degree 3: (Replace it by a triangle)

Hardware and Software Support. The experiments were performed on a Sun UltraSparc CPU with 250 Mhz, running under Solaris 2.5. 2 gigabyte main memory were shared with 13 other CPUs; our own memory usage was always 150 MB or less. In general, we used the SUN Workshop CC compiler with optimization flag -fast. (We also performed an experiment on the influence of different optimization options without seeing significant differences.) The advantage of the multiprocessor machine was reproducibility of the results, as the operating system has no need to interrupt since requests by other processes were delegated to other CPUs.

Experimental Method We used the network described up front. We picked 10,000 arbitrary plans from the case study. We used the timing mechanism provided by the operating system with granularity .01 seconds (1 tick). We performed experiments only if the system load did not exceed the number of available processors, i.e. processors do not get shared. As long as this condition was not violated during the experiment, the running times were fairly consistent, usually within relative errors of 3%.

We used (a subset) of the following values measurable for a single or a specific number of computation to conclude the reported results

- (average) running time excluding i/o
- number of nodes fringe/expanded
- pictures of fringe/expanded nodes
- maximum heap size
- number and length of the path

Software Design We are using the object oriented features as well as the templating mechanism of C++ to easily combine different implementations. We also use preprocessor directives and macros. We do not use virtual methods (even so it is tempting to create a purely virtual "network" base class) to avoid unnecessary function calls (by this enable inlining of functions).

There are classes encapsulating the following elements of the computation:

- network (extensibility and different levels of detail lead to small, linear hierarchy)
- plans: (o, d) pairs and real paths, starting time
- heap
- labeling of the graph and using the heap
- storing the shortest path tree
- Dijkstra's algorithm

As to be expected, this approach leads to a formal overhead of function calls. As it turns out, the compiler optimization can take care of this fairly well. (There is a factor of 2-3 difference in running time between debugging flag and full optimization.)

6 Experimental Results

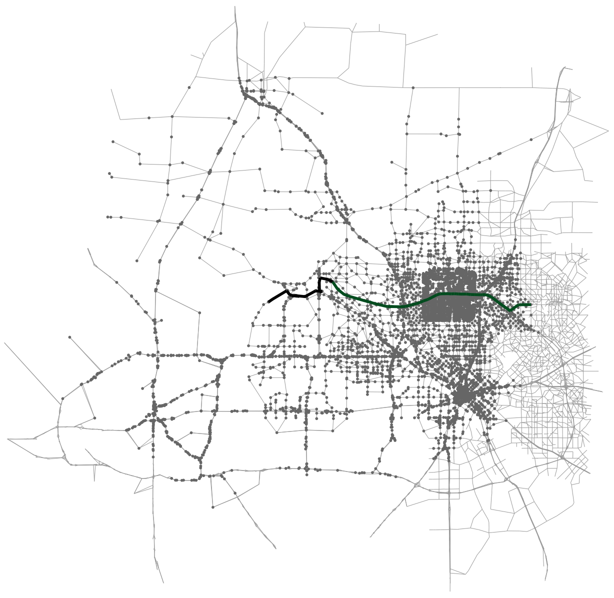
Design Issues about Data Structures We begin with the design decisions regarding the data structures used.

A number of alternative data structures were considered in the hope of investigating if these improvements results in substantial improvement in the running time of the algorithm. The alternatives tested included the following. (i) Arrays versus Heaps , (ii) Deferred Update, (iii) Hash Tables for Storing Graphs, (iv) Smart Label Reset (v) Heap variations, and (vi) struct of arrays vs. array of structs. Appendix contains a more detailed discussion of these issues. We found, that indeed good programming practice, using common sense to avoid unnecessary computation and textbook knowledge on reasonable data structures are useful to get good running times. For the alternatives mentioned above, we did not find substantial improvement in the running time. More precisely, the differences we found were bigger than the unavoidable noise on a multi-user computing environment. Nevertheless, they were all below 10% relative difference. Thus, we do not discuss these results in further detail.

Analysis of results. The plain Dijkstra, using static delays calculated from reported free flow speeds, produced roughly 100 plans per second. Figure 1 illustrates the improvement by the obatined by A^*. The numbers shown in the corner of the network snapshots tell an average (100 repetitions) running time for this particular O-D-pair, (destroying hash effects between subsequent runs) in system ticks. It also gives the number of nodes expanded and fringe nodes. Note the changed scale of the depictions due to the different nodes expanded. Overall we found that A^* is faster than basic Dijkstra's algorithm by roughly a factor of 2. Also, recall that for the original network Sedgewick and Vitter's heuristic was not applicable: it turned out that there exist some links that have reported length much smaller (factor 100) than the Euclidean distance of the endpoints. To be able to conduct any reasonable experiment, we modified ("normalized") the network as reported above: If necessary the reported length was changed to Euclidean distance, to ensure the correct inequality.

Modified A^* (Overdo Heuristic) Next consider the modified A^* algorithm – the heuristic is parameterized by the multiplicative factor used to weigh the Euclidean distance estimate to the desitnation. We call it the *overdo* parameter due to obvious reasons. As a result it is natural to discuss the time/quality trade-off of the heuristic as a function of the *overdo* parameter. Figure 2 summarizes the performance. In the figure the X-axis represents the overdo factor, being varied from 0 to 100 in steps of 1. The Y-axis is used for multiple attributes which we explain below. First, it is used to represent the average running time per plan. For this attribute, the scale is .02 seconds per unit. As depicted by the solid line the average time taken without any overdo at all is 12.9 microseconds per plan. This represents the base measurement (without taking the geometric information into account). Next, for overdo value of 10 and 99 the running times are respectively 2.53 and .308 microseconds. On the otehr hand, the quality of the solution produced by the heuristic detiorates as the overdo factor is increased. We used two quantities to measure the error — the maximum relative error incurred over 10000 plans and the more interestingly the number of plans worse than a given threshold error. The maximum relative error ranges from 0 for overdo factor 0 to 16% for overdo value 99. For the other error measure, we plot one curve for each threshold error of 0%, 1%, 2%, 5%, 10%. The following conclusions can be drawn from our results.

1. The running times improve significantly as the overdo factor is increased. Specifically the improvements are a factor 5 for overdo parameter 10 and almost a factor 40 for overdo parameter 99.


ticks 2.40, #exp 6179, #fr 233

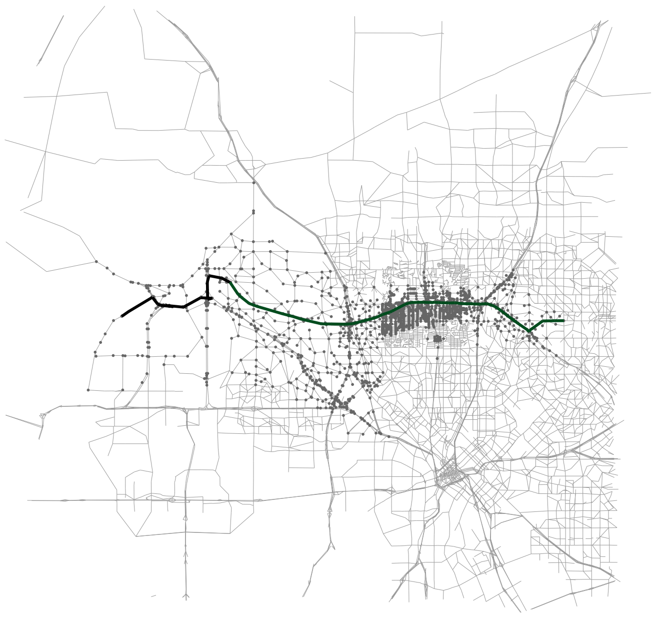

ticks 0.64, #exp 1446, #fr 316

Figure 1: Figure illustrating the number of expanded nodes while running (i) Dijkstra (ii) A^* algorithms. As the figures clearly show the A^* heuristic clearly is much more efficient in terms of the nodes it visits. In both the graphs, the path is outlined as a dark line. The fringe nodes and the expanded nodes are marked as dark spots. The underlying network is shown in light grey.

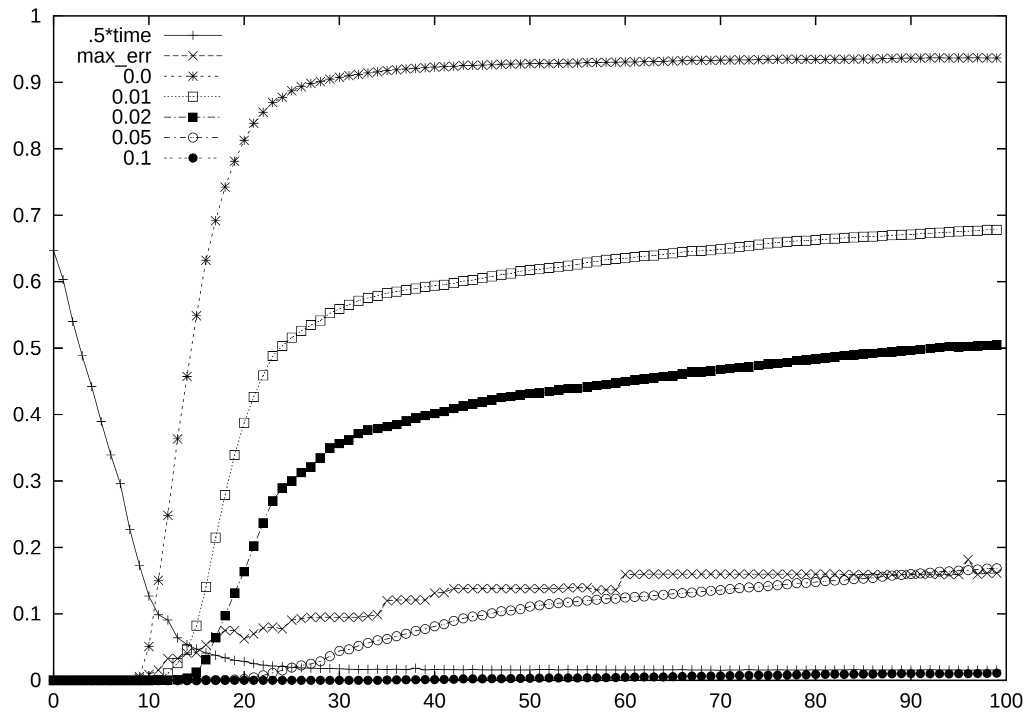


Figure 2: Influence of the "Overdo"-Parameter on running time and quality of paths

2. In contrast, the quality of solution worsens much more slowly. Specifically, the maximum error is no worse than 16% for the maximum overdo factor. Moreover, although the number of erroneous plans is quite high (almost all plans are erroneous for overdo factor of 99), most of them have small relative errors. To illustrate this, note that only around 15% of them have relative error of 5% or more.

3. The experiments and the graphs suggest an "optimal" value of overdo factor for which the running time is significantly improved while the solution quality is not too bad. Thus our experiments are a step in trying to find an empirical time/performance trade-off as a function of the overdo parameter.

4. We also found that the near-optimal paths produced were visually acceptable and represented a feasible alternative route guiding mechanism. This method finds alternative paths that are quite different than ones found by the k-shortest path algorithms and seem more natural. Intuitively, the k-shortest path algorithms, find paths very similar to the overall shortest path, except for a few local changes.

7 Discussion of Results

First, we note that the running times for the plain Dijkstra are reasonable as well as sufficient in the contex of the TRANSIMS project. Quantitatively, this means the following: TRANSIMS is run in iterations between the micro-simulation, and the planner modules, of which the route planner is

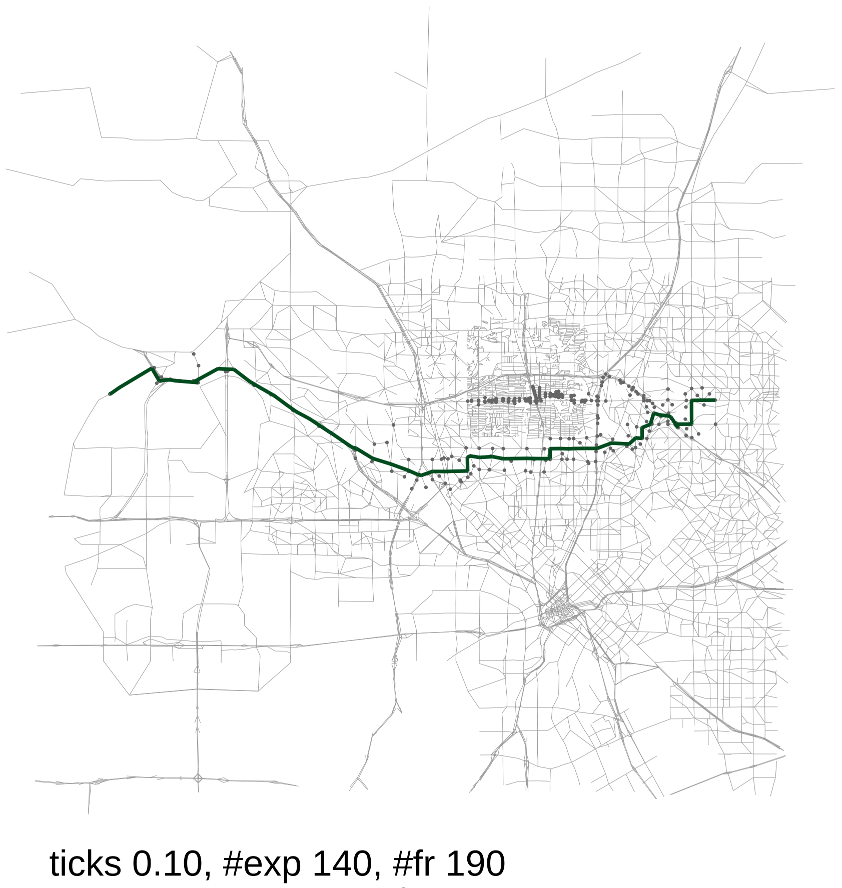


Figure 3: for illustration only: two instances of Dijkstras algorithms with a very high overdo parameter start at origin and destination respectively. One of them really creates the shown path, the beginning of the other path is visible as a “cloud” of expanded nodes

one part. The Portland network we are intending to use has about 120 000 links and about 80 000 nodes. Simulating 24 hours of traffic on this network will take about 24 hours computing time on our 14 CPU machine. There will be about 1.5 million trips on this network. Routing all these trips should take $1.5 \cdot 10^6$ trips $\cdot\, 0.5$ sec/trip ≈ 9 days on a single CPU and thus less than 1 day on our 14 CPU machine. Since re-routing typically concerns only 10% of the population, we would need less than 3 hours of computing time for the re-routing part of one iteration, still significantly less than the micro-simulation needs.

Our results and the necessary contstraints placed by the functionality requirement of the overall system imply that bidirectional version of Dijkstra’s algorithm is not a viable alternative. Two reasons for this are: (i) The algorithm can not be extended in a direct way to path problems in a multi-modal and time dependent networks, and (ii) the running times of A^* is better than the bidirectional variant; the modified A^* is much more faster.

We have recently begun research for the next case study project for TRANSIMS. This case study is going to be done in Portland, Oregon and was chosen to demonstrate the validate our ideas for multi-modal time dependent networks with public transportation following a scheduled movement. Our initial study suggests that we now take .5 seconds per plan as opposed to .01 seconds in the Dallas Ft-Worth case. All these extensions are important from the standpoint of finding algorithms for realistic transportation routing problems. We comment on this in some detail below. Multi-modal networks are an integral part of most MPO’s. Finding optimal (or near-optimal) routes in this environment therefore constitutes a real problem. In the past, solutions for routing in such networks was handled in an adhoc fashion. In [BJM98], we have proposed models and corresponding algorithms to solve such problems.

Next consider another important extension — namely to time dependent networks. In this case the edge length is assumed to be a function of time. We make an important modeling assumption, namely it does not pay a person to wait. This need not be true in general but is adequate for most purposes. This implies that the edge length function is monotonically non-increasing. Time dependent networks can also be used to models public transportation systems with fixed schedules. By using an appropriate extension of the basic Dijkstra's algorithm, one can calculate optimal paths in such networks.

8 Conclusions

The computational results presented in the previous sections demonstrate that Dijkstra's algorithm for finding shortest paths is a viable candidate for compute route plans in a route planning stage of a TRANSIMS like system. In fact, even more interestingly, the results demonstrate that the algorithm that has optimized well compares well (or even sometimes better) than several heuristics proposed in the literature. Thus such an algorithm should be considered even for ITS type projects in which we need to find routes by an on-board vehicle navigation systems.

In the context of the TRANSIMS project, we are faced with the problem of routing many millions of trips in iteration with a micro-simulation. Most trips have entirely different characteristics, such as different starting locations, different starting times, and different preferences towards mode choice. This leads to the consideration of one-to-one shortest path algorithms, as opposed to algorithms that construct the complete shortest-path tree from a given starting (or destination) point. As is well known, the worst-case complexity of one-to-one shortest path algorithms is the same as of one-to-all shortest path algorithms. Yet, in terms of our practical problem, this is not applicable. First, a one-to-one algorithm can stop as soon as the destination is reached, saving computer time especially when trips are short (which often is the case in our setting). Second, since our networks are roughly Euclidean, one can use this fact for heuristics that reduce computation time even more. One heuristic, the Sedgewick-Vitter or A^* algorithm, generates results that are provably optimal, but is a heuristic in the sense that the worst-case complexity does not get any better although practical computing times decrease. One can extend the approach of Sedgewick-Vitter or A^* towards a "true" heuristic where routes are no longer optimal but computation time goes down even more. The above approaches were evaluated in the context of the TRANSIMS Dallas-Fort Worth case study. The underlying road network was a so-called focussed network, with all streets including the local ones in a five times five miles study area, and more and more streets left out when going away from the study area. For that case, SV/A^* turns out to be about a factor of two faster than regular Dijkstra; the second heuristic could save, for example, another factor of 5 while generating results within 1% of the optimal solution.

Making the algorithms time-dependent in all cases slowed down the computation by not more than a factor of two. Since we are using a one-to-one approach, adding extensions that for example include personal preferences (e.g. mode choice) are straightforward; preliminary tests let us expect slow-downs of not more than a factor 30. This apperently mainly induced by an quadrupeled network (splitting links and adding bus topology), complicated time dependency functions representing scheduled busses and presumably most import by the different type of delays inducing a qualtitatively different exploration of the network by the algorithm. Extrapolations of the results for a Portland problem (the next TRANSIMS case study) show that, even when time-dependent and with the extensions, the route planning part of TRANSIMS still uses significantly less computing time than the micro-simulation.

Last, we want to mention that under certain circumstances the one-on-one approach chosen in this paper may also be useful for ITS applications. This would be the case when customers whould require customized route suggestions, so that re-using a shortest path tree from another calculation may no longer be possible.

Acknowledgments: We would like to thank the members of the TRANSIMS team in particular, Doug Anson, Chris Barrett, Richard Beckman, Roger Frye, Terence Kelly, Marcus Rickert, Myron Stein and Patrice Simon for providing the software infrastructure, pointers to related literature and numerous discussions on topics related to the subject. The second author wishes to thank Myron Stein for long discussion on related topics and for his earlier work that motivated this paper. We also thank Joseph Cheriyan, S.S. Ravi, Prabhakar Ragde, R. Ravi and Aravind Srinivasan for constructive comments and pointers to related literature.

References

[AMO93] R. K. Ahuja, T. L. Magnanti and J. B. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[AHU] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison Wesley, Reading MA., 1974.

[AB+97] D. Anson, C. Barrett, D. Kubicek, M. Marathe, K. Nagel, M. Rickert and M. Stein, *Engineering the Route Planner for Dallas Case Study* Technical Report, Los Alamos National Laboratory, Feb 97.

[BJM98] C. Barrett, R. Jacob, M. Marathe, *Formal Language Constrained Path Problems* accepted for SWAT98, Technical Report, Los Alamos National Laboratory, LA-UR 98-1739.

[TR+95a] C. Barrett, K. Birkbigler, L. Smith, V. Loose, R. Beckman, J. Davis, D. Roberts and M. Williams, *An Operational Description of TRANSIMS*, Technical Report, LA-UR-95-2393, Los Alamos National Laboratory, 1995.

[CS97] R. Beckman et. al. *TRANSIMS-Release 1.0 – The Dallas Fort Worth Case Study*, LA-UR-97-4502

[CGR96] B. Cherkassky, A. Goldberg and T. Radzik, *Shortest Path algorithms: Theory and Experimental Evaluation,* Mathematical Programming, Vol. 73, 1996, pp. 129–174.

[GGK84] F. Glover, R. Glover and D. Klingman, *Computational Study of am Improved Shortest Path Algorithm*, Networks, Vol. 14, 1985, pp. 65–73.

[EL82] R. Elliott and M. Lesk, "Route Finding in Street Maps by Computers and People," *Proceedings of the AAAI-82 National Conference on Artificial Intelligence,* Pittsburg, PA, August 1982, pp. 258-261.

[HNR68] P. Hart, N. Nilsson, and B. Raphel, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. on System Science and Cybernetics,* (4), 2, July 1968, pp. 100-107.

[HM95] Highway Research Board, *Highway Capacity Manual*, Special Report 209, National Research Council, Washington, D.C. 1994.

[LR89] M. Luby and P Ragde, "A Bidirectional Shortest Path Algoroithm with Good Average Case Behaviour," *Algorithmica*, 1989, Vol. 4, pp. 551-567.

[Ha92] R. Hassin, “Approximation schemes for the restricted shortest path problem,” *Mathematics of Operations Research*, vol. 17, no. 1, pp. 36-42 (1992).

[Ma] Y. Ma, “A Shortest Path Algorithm with Expected Running time $O(\sqrt{V}\log V)$,” Master’s Thesis, University of California, Berkeley.

[MCN91] J.F. Mondou, T.G. Crainic and S. Nguyen, *Shrotest Path Algorithms: A Computational Study with C Programming Language*, Computers and Operations Research, Vol. 18, 1991, pp. 767–786.

[NB97] K. Nagel and C. Barrett, *Using Microsimulation Feedback for trip Adaptation for Realistic Traffic in Dallas*, International Journal of Modern Physics C, Vol. 8, No. 3, 1997, pp. 505-525.

[Pa74] U. Pape, *Implementation and Efficiency of Moore Algorithm for the Shortest Root Problem*, Mathematical Programming, Vol. 7, 1974, pp. 212–222.

[Pa84] S. Pallottino, *Shortest Path Algorithms: Complexity, Interrelations and New Propositions*, Networks, Vol. 14, 1984, pp. 257–267.

[Po71] I. Pohl, “Bidirectional Searching,” *Machine Intelligence*, No. 6, 1971, pp. 127-140.

[SV86] R. Sedgewick and J. Vitter “Shortest Paths in Euclidean Graphs,” *Algorithmica*, 1986, Vol. 1, No. 1, pp. 31-48.

[SI+97] T. Shibuya, T. Ikeda, H. Imai, S. Nishimura, H. Shimoura and K. Tenmoku, “Finding Realistic Detour by AI Search Techniques,” Transportation Research Board Meeting, Washington D.C. 1997.

[TR+95b] L. Smith, R. Beckman, K. Baggerly, D. Anson and M. Williams, *Overview of TRANSIMS, the TRansportation ANalysis and SIMulation System* Technical Report, LA-UR-95-1641, Los Alamos National Laboratory, May, 1995.

[ZN98] F. B. Zhan and C. Noon, *Shrotest Path Algorithms: An Evaluation using Real Road Networks* Transportation Science, Vol. 32, No. 1, (1998), pp. 65–73.

Appendix: Description of Basic Algorithms

In this section, we describe the basic algorithms considered in this paper. Most of the results in this section are not new; we recall them here for completeness and for description of experimental results.

8.1 Dijkstra's Algorithm

Dijkstra's algorithm solves the single source shortest path problem on a weighted (un)directed graph $G(V, E)$, when all the edge weights are nonnegative. Let $w(u, v)$ denote the weight of an edge in the network.

Suppose we wish to find a shortest path from s to t. Dijkstra's algorithm maintains a set S of vertices whose final shortest paths from the source s have been already computed. The algorithm repeatedly finds a vertex in the set $u \in V - S$ which has the *minimum* shortest path estimate, adds u to S and updates the shortest path estimates of all the neighbors of u that are not in S. The algorithm continues until the terminal vertex is added to S. In general, it is convenient to think of the vertices in the graph being divided into three classes during the execution of the algorithm: (i) *shortest path tree vertices* – (those which have been added to S and hence their shortest path has already been determined, (ii) *unseen vertices*– those for which the distance estimate is ∞ and (iii) *fringe vertices* – those that are adjacent to the vertices in S but have themselves not been added to S. Now each iteration of the algorithm consists of adding a fringe vertex with minimum distance to the shortest path tree and updating its neighbors to be fringe vertices. Using this terminology, initially, only s is a shortest path tree vertex, neighbors of s are fringe vertices, and others are unseen vertices.

DIJKSTRA'S ALGORITHM outlines the steps of the algorithm. In the remainder of the section, we will use $\delta(u)$ to denote the cost of a shortest path from s to u. We will also assume that $|V| = n$ and $|E| = m$. Also, for a given vertex v let $N(v)$ denote the set of neighbors of v i.e. $N(v) = \{w \mid (v, w) \in E\}$. Finally, by the phrase *extract a vertex* from V we mean choose a vertex and delete it from V.

DIJKSTRA'S ALGORITHM:

- *Input:* $G(V, E)$ - a network, a source s and a destination vertex d and a non-negative weight function $l : E \rightarrow Z^+$.
- 1. *Initialization:* Set $S = \phi$, $d(s) = 0$ and $\forall v \in V - \{s\}, d(v) = \infty$. Found = 0.
  2. *Iterative Step:* **while** Found = 0 **do**
     (a) *Extract Minimum Step:* Among all vertices $v \in V - S$ extract a vertex v with minimum value of $d(v)$. Set $S = S \cup \{v\}$. If $v = d$ then set Found = 1.
     (b) *Decrease (Update) Key:* For each edge (v, w), such that $w \in N(v)$, set $d(w) = \min\{d(w), d(v) + w(v, w)\}$.
- *Output:* A shortest path from s to d, i.e. a path $p =< v_0, \ldots v_k >$ where $v_0 = s$ and $v_k = d$ and the weight $w(p) \sum_{i=1}^{i=k} w(v_{i-1}, v_i)$ is the minimum over all paths from s to d

8.2 Bidirectional Dijkstra's Algorithm

The bidirectional algorithm has been used in the operations research community and analyzed by theoretical computer scientists providing quantitative reasons for its improved performance. (See [LR89, Ma] for more details.) The bidirectional search algorithm consists of two phases. In the first phase we alternate between two unidirectional searches: one forward from s, growing a tree spanning a set of nodes S for which the minimum distance from s is known, and the second that consists of growing a tree spanning a set of nodes D for which the minimum distance from d is known. We alternately add one node to S and one to D until an edge crossing from S to D is drawn. At this point, the shortest path is known to lie within the search trees associate with S and D except for one additional edge from S to D. A geometric interpretation of the algorithm (in which the edges have unit weights) is as follows:

> We start growing a ball around s and t, at each time step, the ball grows by 1 unit (in terms of radius). We stop the algorithm, when the two balls collide; i.e. there exists a vertex that becomes a part of both the balls. The path $s \sim\sim v \sim\sim t$, where v is the vertex where the balls collide, represents the shortest path from s to t.

As mentioned earlier, in case of weighted graph we also need to consider one extra cross for the possible inclusion in the shortest path.

Lemma 8.1 *The following statements hold:*
(1) The shortest path from s to t has at most one cross edge from a vertex in S to a vertex in D.
(2) $w(s-k-t) \leq 2w(s-t)$

Proof Sketch: We denote $w(s-k-t)$ as the weight of the shortest path from s to t going through k. Let $P = s \sim\sim x - \alpha - y \sim\sim t$ denote a shortest path that has more than one edge. The following inequalities are immediate from the correctness of Dijkstra's algorithm and the fact that P is a shortest path: $w(s-x)+w(x-\alpha) \geq w(s,k)$ and $w(t-y)+w(y-\alpha) \geq w(y,k)$. This implies $w(s-x)+w(x-\alpha)+w(t-y)+w(y-\alpha) \geq w(s,k)+w(y,k)$ which is a contradiction.
Part 2: Let us say that shortest path from s to t is of the form $s \sim\sim x - y \sim\sim t$, where $x \in S$ and $y \in D$. The following inequalities are immediate: $w(s,k) \leq w(s-x)+w(x-y)$; $w(y,k) \leq w(t-y)+w(y-x)$. This implies $w(s,k)+w(k,y) \leq w(s-x)+w(x-y)+w(t-y)+w(y-x) \leq 2w(s-x-y-t)$.

In [LR89], the authors show that if the weight of each edge in a complete directed graph with n nodes is chosen from an exponential distribution, with high probability the bidirectional search terminates after examining a substantially fewer edges than the unidirectional search.

Theorem 8.2 *Let a, b, c be constants. Define a family of probability distributions over a n-node directed graph with edge weights, one distribution for for each n. each edge (i, j) has a probability of $(a \log n)/b$ of being present. For those edges that are chosen, the length of the edge l_e is independently chosen according to a probability distribution whose density function f_e has a value bounded between b and c. The expected time to find a $s-t$ shortest path using bidirectional search is $O(\sqrt{n})$.*

8.3 A Modification For Euclidean Graphs: A^*-Algorithm

When the underlying network is Euclidean, it is possible to improve the average case performance of Dijkstra's algorithm. Euclidean graphs are defined as follows. The vertices of the graph correspond to points in R^d and the weight of each edge is proportional to the Euclidean distance between the two points. Typically, while solving problems on such graphs, the inherent geometric information is ignored by the classical path finding algorithms. The basic idea behind improving the performance of Dijkstra's algorithm is from Sedgewick and Vitter [SV86] and is originally attributed to Hart Nilsson and Raphel [HNR68] is simple and can be described as follows. To build a shortest path from s to t, we use the original distance estimate for the fringe vertex such as x, i.e. from s to x (as before) *plus* the Euclidean distance from x to t. Thus we use global information about the graph to guide our search for shortest path from s to t. To formalize this, define $D(x, y)$ to be the Euclidean distance between x and y and define $l(x, y)$ to be the shortest path from x to y in the graph. The length of the path as usual is equal to the sum of the edge lengths that constitute the path: the weight of an edge (x, y) is defined to be $D(x, y)$. Now each fringe vertex x is assigned the following value: $\min_w\{l(s, w) + D(w, x)\} + D(x, t)$ The resulting algorithm runs much faster than Dijkstra's algorithm on typical graphs for the following reasons: (i) The shortest path tree grows in the direction of t and (ii) The search of the shortest path can be terminated as soon as t is added to the to the shortest path tree. The correctness of the algorithm follows from the fact that $D(x, t)$ is a lower bound on $l(x, t)$. Another way to interpret the algorithm and its correctness is by using the concept of vertex potentials – an idea first used by Gabow.

Concept of Vertex Potentials. Each vertex is assigned a non-negative value $D(x)$ – called its *potential.* The intuition is that when you enter a vertex v we receive $D(v)$ dollars which are deducted from the path and when we leave a vertex we add that amount of money to the path. Using these potentials, let us define the length of the edges as follows

$$\forall (u, v) \in E,\ \ \tilde{l}(u, v) = l(u, v) + D(u) - D(v)$$

The potentials are called *admissible* or *feasible* if the new lengths are all positive. The following theorem shows that the the shortest paths in the graph with modified weights remains the same.

Theorem 8.3 *Let D be a set of admissible vertex potential. Then the weight of a path $p =< s = v_1, \dots v_r = t >$ from s to t is given by*

$$\tilde{w}(p) = \sum_{i=1}^{i=n} l(v_i v_{i+1}) + D(s) - D(t)$$

In other words the length of each path from s to t is changed by the same constant additive factor. Thus if p is a shortest $s - t$ path in the original graph then it is still the shortest path in the graph with modified edge weights.

Proof: Consider a path $p =< s = v_1, \dots v_r = t >$. Clearly its weight under modified weight function is

$$\tilde{w}(p) \quad = \quad \sum_{i=1}^{i=n} \tilde{l}(v_i v_{i+1})$$

$$\begin{aligned} &= \sum_{i=1}^{i=n} l(v_i v_{i+1}) + D(v_i) - D(v_{i+1}) \\ &= \sum_{i=1}^{i=n} l(v_i v_{i+1}) + D(s) - D(t) \\ \tilde{w}(p) &= w(p) + D(s) - D(t) \end{aligned}$$

8.4 Modified A^*

We briefly discuss some of the heuristic improvements to the basic A^* algorithm that can be used in practice. Again, recall that in many practical situations (including TRANSIMS), it is not necessary to find exact shortest paths – approximately shortest paths suffice. We tried two heuristic solutions in this context.

(1) The modified A^* algorithm. Recall that the current label of a vertex consists of two components — its shortest distance to the source and an estimate (usually the Euclidean distance) to the destination. By choosing an appropriate multiplicative factor, we can increase the contribution of the second component in calculating the label of a vertex. From a intuitive standpoint this corresponds to giving the destination a high potential, in effect biasing the search towards the destination. The resulting paths clearly need not be optimal. By choosing the appropriate bias factor, one can typically get faster algorithms at the cost of accuracy. As our results in Section 6 point out, it appears that an appropriate constant results in a very good trade-off between quality of solution and the time required.

(2) Combining A^* with Bidirectional Search The discussion in the above sections suggests combining the bidirectional search heuristic with the A^* search. One possible way to do it is to use two potentials $D_s(u)$ and $D_t(u)$ for each vertex, the potentials reflecting the lower bounds (usually geometric distances) of u from s and t. A naive of implementing this idea is unfortunately incorrect, since the two potentials imply building shortest path trees from s and t. As shown in [SI+97], a modified potential suffices to ensure the correctness of the algorithm.

9 Discussion on Data Structures

(1) Arrays versus Heaps. In a naive implementation of the algorithm using an array $\mathcal{A}$, in which for each vertex, v_i we store the value of $d(v_i)$ in location $\mathcal{A}(i)$. In each iteration *Extract Minimum Key* takes $O(n)$ time (finding a minimum value in an unsorted array takes $O(n)$ time) and *Decrease Key* takes time $O(deg(v))$. Here $deg(v)$ denotes the degree of v. The total running time is therefore $\sum_v O(n + deg(v)) = O(n^2 + m)$. Using *Binary Heaps* (as has been done in the current implementation of the algorithm), we can improve the running time. First consider *Extract Minimum Key* operation. The time to do this is $O(\log n)$ since we simply pick the top of the heap and then process the data structure to maintain the heap property (using HEAPIFY). Next consider the *Decrease Key* operation. This operation takes time $O(deg(v) \log n)$ for the following reason. We need to update the distance estimate for each of the $deg(v)$ neighbors, each operation taking time $O(\log n)$. The time to build the heap for the first time is $O(n)$. Thus the total running time of the algorithm is $\sum_v O((\log n) + deg(v)\log n)) = O(n\log n + m\log n) = O(n+m)\log n$. We also considered using Fibonacci Heaps. Our experimental analysis revealed that typically the number of nodes that are kept in a heap is around 500; thus using a more sophisticated data structure with higher constants was not likely to yield better results in practice. Using Fibonacci heaps could potentially improve the theoretical running time of the algorithm by of $\log(\mathcal{H})$, where $\mathcal{H}$ denotes the maximum heap size at any stage of the execution. This implies an improvement of at most a factor of 9. But the constants with the heap operations and the complicated code for implementing this data structure weigh more heavily against it.

(2) Deferred Update. Recall that we need to update the values of the distance estimates in Step 2b of DIJKSTRA'S ALGORITHM. Assume that the heap is $\mathcal{H}$, and degree of a node v being d_v, it would take roughly $2d_v \log \mathcal{H}$ operations to update the distance estimates. The reason for this is as follows: We can maintain an auxiliary array that keeps pointers to the nodes in the heap. Every time a nodes distance estimate is updated, the node moves through the heap (as a part of HEAPIFY operation) to settle in the final position. During the course of this other nodes on its path also change positions. This implies that the pointed values for each of the nodes need be updated. (We are assuming an array implementation of the Heap.) Another possible way to do this is to insert multiple copies of a node in the heap. In this way, the time taken is roughly proportional to adding these nodes plus the additional factor depending on the size of the heap for future operations. Again, let d_v denote the degree of a node and d_{max} be the maximum degree. Then the heap size grows at most by a multiplicative factor of d_{max}. Since the Heap operations take time roughly $\log \mathcal{H}$ this implies that the total time for executing Step 2b is no more than $d_{max}\log(d_{max}\mathcal{H})$ which is $d_{max}(\log \mathcal{H} + \log d_{max})$. Typically, the average degree of a node in the Case study network is $2.8 \sim 3$ and you expect that it only gets inserted roughly only by half its neighbors resulting in an average increase of no more than 4 on the size of the heap. This implies that we spend only an additional additive factor of $2d_v$ for each run of Step 2b.

(3) Hash Tables for Storing Graphs. The graph or the input we received from Dallas MPO consists of long Link and Node Id's. Although the naming convention is useful for in other contexts, such a naming convention yields a inefficient use of the domain space. To illustrate the point, the link and the node It's given were typically made of 32 bits long. Thus the name space of for the nodes is roughly 2^{32}. In contrast the number of nodes is roughly $10^4 \sim 2^{12}$. Such a discrepancy immediately motivated a use of hash tables to improve the naming space utilization. We used a Hash Table of size roughly 2^{14} (i.e. address is bits long). One important reason for doing this is clearly the efficiency gained during accessing the long names. The efficiency is obtained for two possible reasons. The first and more important reason is that the array used to store the structure (information) associated with each node is small enough to typically fit the first level cache. In contrast arrays of size 2^{32} will never a fast cache and thus will imply a significant increase in the processing time. It is well known that memory access is significant bottleneck in the design of fast algorithms. Another small reason is that small words might be useful in efficient access of memory contents. Also, note that the Hash table needs to be accessed only during input and output of the plans and thus the process is not expensive.

(4) Smart Label Reset We now discuss the improvement performed in the context of finding paths for a number of travelers. Note that in Step 1 we need to set the distance estimates of all the nodes to be initialized to infinity. This takes $O(n)$ time per run of the algorithm. We instead relabel only those nodes whose labels have changed during the course of the algorithm. This simply consists of the nodes that were at anytime inserted in the heap. Since on an average the total number of nodes visited is a small fraction of the total number of nodes (in fact is $O(\sqrt{n})$ for bidirectional implementation) this yields significant improvements in the running time of the algorithm.

(5) Heap tricks At the innermost loop of our heap implementation are two small details: one is the test on a special case at the end of the heap. This test can be replaced by setting unused elements of the array to the value infinity, by this replacing an operation in the loop by (possibly) one more iteration in the loop. The other possibility is to "streamline" the comparison at this loop from possibly four down to three.

(6) struct of arrays vs. array of structs Following object oriented design goals one ends up having different, independent arrays for storing data for the network, label-setting and the shortest-path-tree module. Considering caching behavior of the processor it seems advantageous to combine these to one big array of structs having entries for the different modules.